DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming

Organización del Computador II

Primer Cuatrimestre de 2018

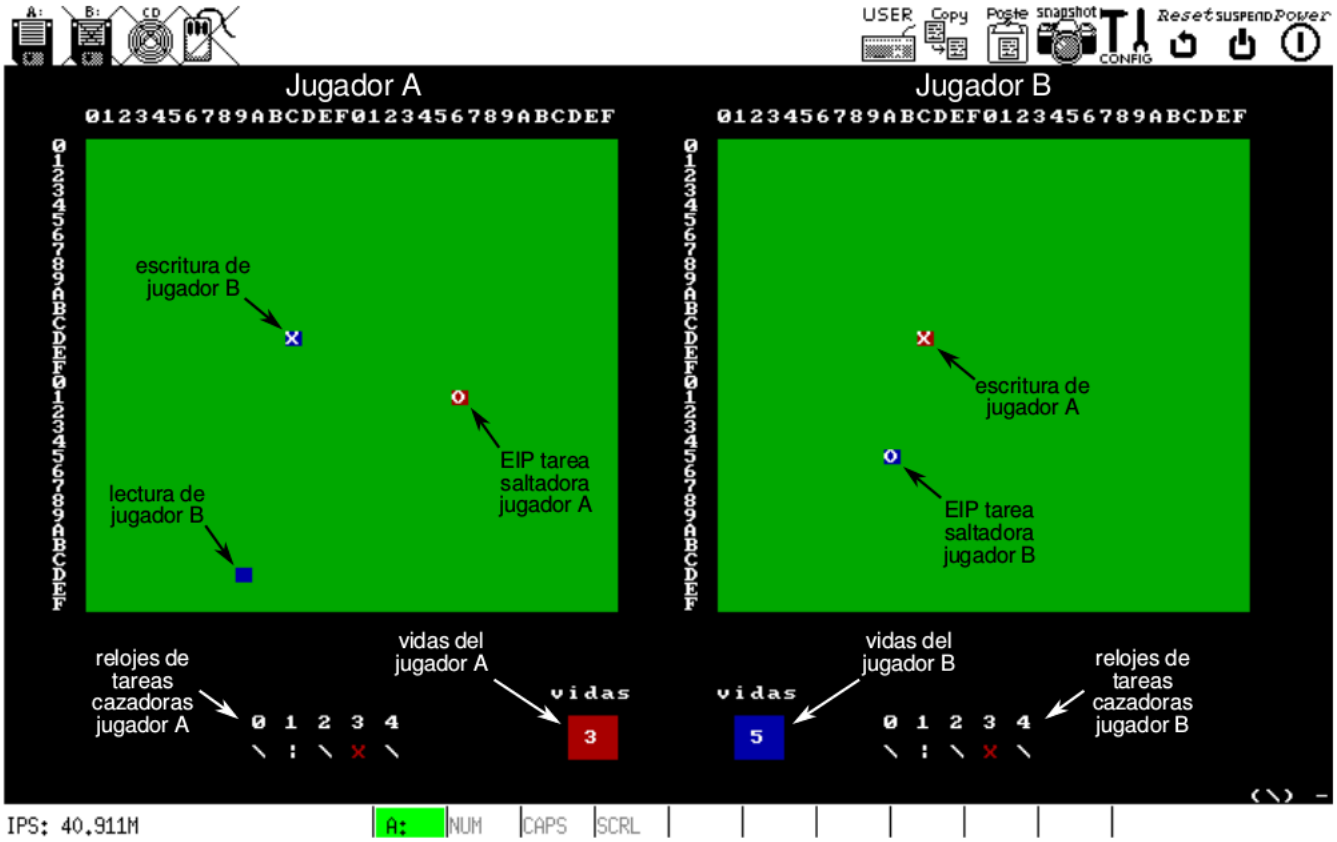
Integrante	LU	Correo electrónico
Castro Luis	422/14	castroluis1694@gmail.com
González Paula	774/15	paula.cgs@hotmail.com
Imperiale Luca	436/15	luca.imperiale95@gmail.com
Naselli Matias	82/15	matiasnaselli@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este proyecto nosotros vamos a aplicar los conceptos de *system programming* vistos en clase. El objetivo es crear un sistema mínimo que soporte unas 10 tareas a nivel usuario. Existen dos tipos de tareas, las tareas cazadoras y las tareas saltadoras. Las tareas cazadoras se encargan de leer y escribir código sobre las tareas saltadoras con el objetivo de eliminarlas.



Esta es una representación de una de las formas en las cual se puede ver el sistema. El objetivo del jugador es hacer que sus tareas cazadoras logren capturarlas o eliminarlas. Cada jugador tiene un total de 5 vidas, el primero en hacer que el otro jugador llegue a 0 vidas gana.

2. Ejercicios

2.1. Ejercicio 1

Para este primer inciso creamos 4 segmentos, dos de código(nivel 0 y 3) y otros dos de datos(nivel 0 y 3 igualmente). Para esto usamos el struct gdt_entry proporcionado por la cátedra. Los limites de los segmentos fueron configurados para que direccionen hasta 314MB(o sea de 0x00000 a 0X13FFF en hexa). Estos segmentos tienen una base que apunta a la dirección 0x0000 siguiendo los esquemas mostrados en el enunciado. El type de estos 4 segmentos es 0x0A para código y 0x02 para datos debido a que esa es la configuración que buscamos para que podamos ejecutar/leer y leer/escribir respectivamente. El bit S(system bit), este bit lo dejamos activo porque es la convención para los segmentos de código/datos. Los bits de DPL son marcados con 0x00 o 0x03 dependiendo del nivel de privilegio del segmento. El bit P esta marcado en 0x01 debido a que queremos que nuestros segmentos esten en la ram y no en la memoria virtual. Para la configuración de AVL decidimos dejar sus bits en ceros debido ya que no los vamos a utilizar. Los bits de L y DB están ligados, si o están ambos en 0x00(indicando que es un segmento de 16bits) o alguno de los dos esta en uno(en el caso de DB significaría que esta en 32bits y en el caso de L es que esta en 64bits), los segmentos corren 32bits. Por ultimo, la granularidad vale 0x01 ya que el máximo offset supera el limite. Para entrar a modo protegido nosotros habilitamos la linea A20(cuya implementacion se nos fue dada por la cátedra), esta linea sirve para poder acceder al área de memoria

alta en modo real, una vez hecho esto cargamos los descriptores de la GDT(usamos la función LGDT, esta para cargar las gdts toma el tamaño de la GDT y su dirección). Después activamos el bit PE de cr0, este registro tiene distintos controles de frags sobre sus bits, el bit PE(Protected Mode Enable) es el que indica que vamos a entrar a modo protegido. Para terminar de entrar a modo protegido hacemos un jump far para cambiar de segmento. Una vez dentro de modo protegido establecemos los selectores de segmento y hacemos que los registros esp y ebp apunten a la dirección 0x27000 para establece donde va a estar la base de la pila. Este segmento se llama GDT_IDX_VIDEO, donde el limite es 0x0fff(este ya que nuestro pantalla tiene dimensión 80*50, o sea 4000 y decidimos redondear para arriba), la base es Ox0B8000, tiene el bit de sistema en 1, el bit de presente en 1, es un segmento de tipo de datos y tiene dpl de nivel 0. Este segmento maneja la memoria destinada a la pantalla. Para iniciar pantalla usamos las funciones definidas por la cátedra screen_drawBox y print_hex, la primera toma un intervalo una letra y un atributo(color de la letra y el color fondo de la letra), este atributo al tener un mismo color de letra con respecto a su fondo produce un efecto de pintado, al hacer esto en toda el área pintamos primero toda la pantalla de negro y luego pintamos de verde 2 cuadrados disjuntos de 32x32 llamando otra vez a screen_drawBox. Para imprimir las letras hexadecimales implementamos las funciones imprimir_hexas_horizontal e imprimir_hexas_vertical, estas dos llaman a la función print_hex e imprimen los números hexadecimales en orden vertical u horizontal dependiendo de la función. Para finalizar utilizamos la función print y draw_screen_box para poder marcar cuantas vidas tiene cada jugador y el estado inicial de sus tareas.

2.2. Ejercicio 2

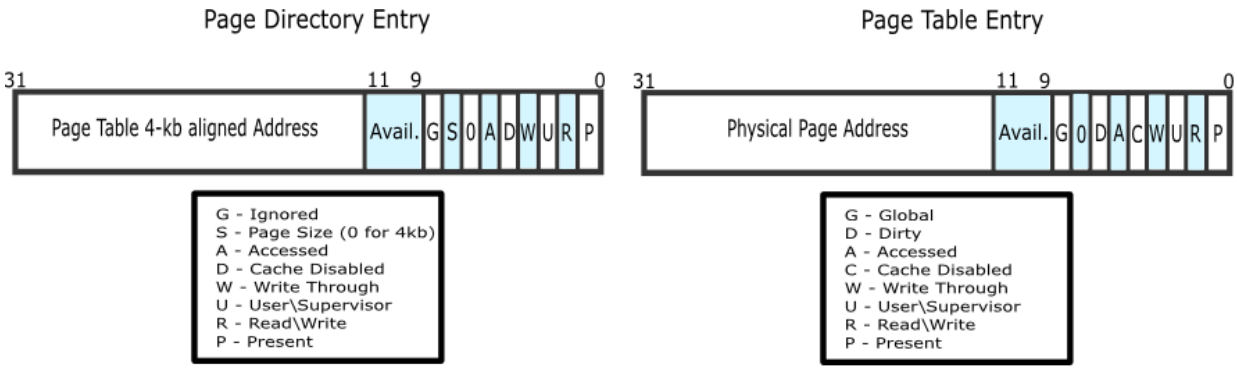
Completamos la función idt_inicializar para que llame a las instrucciones idt_entry de cada interrupción que queremos definir. En idt_entry vamos a configurar el nivel de código se va a ejecutar(nivel 0 en este caso), el offset(numero de interrupción que ocurrió) y el atributo attr cuyo valor sale del DPL que provoco la interrupción. En esta función también esta la macro isr, definida por la cátedra, la cual modificamos para que modificamos para que haga un casteo con el numero de interrupción recibida y la transforme a char para poder imprimirla después por pantalla.

2.3. Ejercicio 3

Para este ejercicio tuvimos que crear primero las entradas en la IDT para el reloj(Interrupción 32), el teclado(Interrupción 33) y la interrupción de software 0x66. Tanto para crear la interrupción del reloj como para el teclado utilizamos la misma función que para las otras interrupciones, llamada IDT_ENTRY, donde las cargamos e inicializamos de manera genérica en idt_entry(como se menciono en el punto 2). Para la interrupción de software, tuvimos que crear un inicializador particular ya que tiene un atributo distinto. Estas interrupciones próximamente van a ser modificadas para cumplir con otras funcionalidades pedidas por otros ejercicios(modo debug, tareas, etc).

2.4. Ejercicio 4

Antes de explicar como hicimos la paginación veamos las estructuras con las que vamos a trabajar. Así es como definimos nuestras structs:



Para empezar hacemos que el directorio de paginas del kernel apunte a la dirección 0x27000 y configuramos el directorio diciendo que este le pertenece al kernel con los bits u_s , este se encuentra en ram con el bit P , es de lectura/escritura y su tabla en la posición 0x28000 según las indicaciones del mapa de memorias del enunciado.

Una vez configurada la primer pagina llamamos identity mapping, este hace K mappeos(K siendo la cantidad de paginas que se van a mappear de 0x0000 a 0x3FFF) tomando de entrada la dirección virtual, la dirección del directorio de pagina, la dirección física a la cual se va a mappear y un char que indica si pertenece al usuario o al sistema. Dentro de mappear pagina, vamos a conseguir tanto el indice del directorio de paginas como el indice del directorio de tablas a partir de la dirección virtual(haciendo los shifteos necesarios), nos fijamos si el directorio de pagina ya esta cargado, si ya esta cargado no hay necesidad de configurarlo, pero en el caso de que no, entonces activamos el byte P para decir que ahora esta en memoria, los bits de u_s se ponen en 0 ya que la pagina le pertenece al kernel, hacemos eso y pedimos una nueva pagina libre del kernel(en este caso es indiferente de donde pedimos la pagina). Para terminar tomamos la tabla a la que apunta la base del directorio de pagina(para eso hacemos un shifteo de 12 bits sobre la base), hacemos que la base de la tabla apunte a los 12bits mas significativos de la dirección física y configuramos sus bits de present, read/write y usuario/supervisor. Para finalizar hacemos la operación lógica or con cr0 y 0x80000000, esto prende el bit PG que activa la paginación.

2.5. Ejercicio 5

Para iniciar la mmu creamos dos variables globales , al inicializar la mmu hacemos que estas apunten al inicio de paginas libres de kernel y de tareas que según el mapa de memoria están en las posiciones de memoria 0x00100000 y el 0x00400000 respectivamente. Ahora para iniciar el directorio de tareas pedimos el indice de tareas, a partir de ahí pedimos una pagina libre del kernel y le damos esa dirección al cr3 de la tarea. Una vez hecho eso prendemos los bits present, read/write y user/supervisor , después pedimos una pagina libre de kernel para la tabla y hacemos que el directorio de paginas tenga de base a la dirección física de la tabla, aplicamos identity mapping sobre el directorio de paginas para poder tener todas las direcciones físicas disponibles. Por enunciado ahora vamos a pedir una pagina libre de tareas y la mappeamos a la dirección 0x08000000(128MB). Con la dirección física de la pagina libre de tareas que pedimos ahora mappeamos solo esa posición de memoria haciendo identity mapping y el cr3 actual, después copiamos el código del indice de la tarea que llamó a la función a la pagina libre de tareas, una vez copiado todo desmappeamos esa pagina y devolvemos el cr3 de la tarea.
Para mappear una pagina en particular pedimos una dirección física, una virtual, la dirección del directorio de paginas y un char que indica si le pertenece supervisor o usuario. Nos fijamos si el directorio de paginas esta cargado(con el bit present), si no lo esta pedimos una pagina libre de kernel, decimos que vamos a leer y escribir y hacemos que el bit u/s valga lo mismo que el del parámetro de entrada. Con la base del directorio de paginas(sacado a partir de la dirección virtual shifteado 12 a izquierda) conseguimos la tabla y hacemos que su base apunte a la dirección física y configure sus atributos para ser próximamente accedida.
Para desmappeo de una pagina se accede al directorio de paginas a través de la dirección virtual y con esta se consigue la tabla, una vez tengamos la tabla ponemos el bit present en 0 para decir que esa tabla tiene basura.

2.6. Ejercicio 6

Primero tuvimos que crear en la gdt 11 descriptores de tss para las tareas inicial, idle, dos saltadoras y 8 cazadoras. Estos descriptores van a ser iguales, con excepción de los 32 bits de base, en donde estará la dirección de memoria donde se encuentre almacenada cada tss. El limite de los descriptores sera 0x067, que es lo que ocupa una tss, su type es 0x09 que indica que es de tipo ejecutable accedido, el bit accedido indica si se ha modificado al segmento desde la última vez que el sistema operativo o ejecutivo despejó el bit. El bit presente va a estar prendido ya que queremos que las tareas estén presentes en la ram, el dpl de la tarea va a ser 3 ya que es el nivel de privilegio de las tareas.

Luego, completamos las tss de cada tarea. Para la inicial no fue necesario inicializar ningún campo, ya que esta solo se usa para poder almacenar el contexto actual en el primer salto a la idle. En la idle, pusimos como indicaba el enunciado el eip en 0x14000, los segmentos de código y datos de nivel 0, el mismo cr3 que el kernel, y la pila en la dirección 0x27000. El resto de los campos van a inicializarse en 0, excepto los eflags en 0x202.

A las saltadoras y cazadoras en cambio, les pusimos descriptores de segmento de nivel 3, ya que correrán en modo user, sus eip empezaran en 0x800000, la pila de nivel 3 en 0x8001000 y eflags en 0x202. El cr3 va a variar de tarea en tarea, pero todos van a estar en el área libre de kernel. Para pila de nivel 0, pedimos una pagina nueva en el área libre de kernel y se la asignamos a cada tarea, y pusimos en su descriptor de segmento el de datos de nivel 0. El resto de los campos van a ser 0.

Con todas estas estructuras ya creadas, hicimos un jmp far a la tarea idle desde el kernel.

2.7. Ejercicio 7

Para el scheduler decidimos tener una estructura con un campo para la tarea actual, un vector de tareas desalojadas y un arreglo del estado de los relojes de las tareas. Los relojes los iniciamos en su posición inicial, y los otros dos campos en valores sin significado, que solo servirán al sistema para el inicio.

Para hacer la función sched_proximoIndice(), lo que usamos fue un arreglo en el cual estaba escrito el orden de las tareas(primero saltadoras, después alternándose cazadoras). Lo que hace la función es usar como indice el campo $tarea\_actual$ del scheduler, sumándole uno, en dicho arreglo. Luego con este valor se arma el descriptor de tss.

Esta función fue utilizada en la interrupción de reloj. Cada vez que se produce, la llamamos y hacemos un jmp far al descriptor de tss que nos devuelve. Internamente esta función chequea que la tarea que esta devolviendo no pertenezca al vector de tareas desalojadas.

Para hacer el desalojo de tareas ante cualquier excepción, primero nos fijamos si la tarea actual en el scheduler es cazadora o saltadora. En el caso de las cazadoras, simplemente la agregamos al vector de tareas desalojadas y hacemos un jmp far a la idle. Por como funciona nuestro scheduler, podemos asegurarnos que no vamos a volver a esa tarea, así que el contexto que guardamos no interesa. En el caso que sea una saltadora, vamos a tener que hacer algunas cosas mas. Lo primero es restarle una vida en el juego, y chequear que este no termine. Necesitaremos que la memoria de la tarea sea reseteada, para esto desmapeamos la pagina que contiene el código, pedimos una nueva pagina en el área libre de tareas, y copiamos el código original ahí. Esto nos asegura que la tarea cuando corra lo va a hacer como la primera vez. Por ultimo, hacemos un jmp far a la tarea Idle, así se completa el quantum, pero antes cambiamos en la pila la dirección de retorno y el esp almacenados a sus valores iniciales, ya que cuando la tarea se ejecute de nuevo va a volver a este contexto, y queremos que con un iret vuelva a correr como si fuese la primera vez.

En la rutina de atención a la interrupción 0x66, lo primero que hicimos fue chequear los parámetros de entrada. Si a la interrupción la llamo una tarea Saltadora, o los valores de offset no son validos, la tarea caerá en una excepción. Si las entradas son validas, chequeamos en eax a que servicio se llamo. En el caso del de número, simplemente devolveremos el valor actual de la tarea actual del scheduler. Para los otros dos casos, vamos a tener que leer o escribir en memoria que no tenemos mapeada, ya que ”pertenece.ª otra tarea. Para esto, primero conseguimos el directorio de tablas de pagina de la tarea a leer o escribir, y, usando el offset y que las tareas solo tienen una pagina, podemos averiguar la dirección física del dato. Luego, mapeamos esa dirección física a alguna pagina distinta a la que usamos (una dirección

que no sea 0x8000000), y con esa dirección lineal finalmente accedemos al dato, ya sea para leerlo o escribirlo.

2.7.1. Modo Debug:

Para el modo debug definimos dos variables, debugMode y debugOpen. La primera se usa para verificar si el modo debug esta activado. Su valor cambia cuando se presiona la tecla ”z”mientras el juego está andando (de activado a desactivado y viceversa). Este valor es necesario para que el sistema sepa si debe mostrarse el valor de los registros cuando se produzca una excepción. Para mostrar la información de debug en pantalla se llama a una función llamada debug definida en C desde la rutinas de atención a las excepciones. Para que a esta función debug no le llegue el error code cómo parámetro, éste (de estar presente en la pila) se popea al comienzo de la rutina. Luego, se verifica si el modo debug está activado para llamar a la función definida en c en caso afirmativo. Como el enunciado indica que el juego debe pararse mientras se muestra la información de debug, lo primero que verifica la rutina de atención de reloj es si debugOpen está activado. En ese caso, la rutina no hace nada como forma de tener el juego interrumpido. Para desactivar debugOpen también se usa la tecla ”z”. Notar que presionar esta tecla mientras el recuadro de debug está activo solo implica reanudar el juego pero no desactiva el modo debug. Para desactivar el modo debug presionando la tecla ”z”, el juego debe estar activo (En pantalla se muestra un mensaje cuando el modo debug está activo).